PROW: A STEP TOWARD AUTOMATIC PROGRAM WRITING

Richard J. Waldinger
Carnegie-Mellon Univ., Pittsburgh, Pa.
Richard C. T, Lee
National Institutes of Health
Bethesda, Md.

Summary

This paper aescriDes a program, called "PRUW", which writes programs. PROW accepts the specification of the program in the language of predicate calculus, decides the algorithm for the program and then produces a LISP program which is an implementation of the algorithm. Since the construction of the algorithm is obtained by formal theorem-proving techniques, the programs that PROW writes are free from logical errors and do not have to be debugged. The user of PROW can make PROW write programs in languages other than LISP by modifying the part of PROW that translates an algorithm to a LISP program. Thus PROW can be modified to write programs in any language. In the end of this paper, it is shown that PROW can also be used as a question-answering program.

Section 1: Introduction

A programmer can easily make two kinds of mistakes. First, the grammar of the programming language that he uses is so complex that he easily makes syntactical errors. Second, it is not easy to always think logically, so he Is liable to make mistakes in constructing algorithms, thus producing logical errors.

Much effort has been made in simplifying the programming languages so that programmers will make fewer syntactical errors. Although one can still complain that the advanced languages available today are still not simple and natural enough, he should find comfort in noting that they are much easier than machine language.

As yet, not much has been done to overcome the second difficulty mentioned above. In this paper, the authors will describe a program called "PROW" (Program Writing), which is designed for this purpose. PROW resembles the "state description" portion of the heuristic compiler (7) by Simon and is much in the spirit of Slagle (8), Raphael and Green (I) in fully using the power of formal logic.

In PROW, the instructions of LISP are axiomatized and stored as axioms. The user expresses the relationship between the input and output variables as a well-formed formula in the first order predicate calculus (2). Suppose the Input and output variables of the program are $X_1, X_2, \ldots, X_m$ and $Y_1, Y_2, \ldots, Y_n$ respectively and that the relation between them is $R(X_1, \ldots, X_m, Y_1, \ldots, Y_n)$. PRCW then constructs the theorem:

$$(X_1)(X_2) \ldots (X_m)(EY_2) \ldots (EY_n)$$
$$R(X_1, \ldots, X_m, Y_1, \ldots, Y_n).$$

PROW first calls a theorem prover to prove the theorem, thus establishing the existence of the program. The output of the theorem prover is a "proof" of the theorem. PROW then calls another program to process the proof and gives the desired program.

Since PROW uses theorem prover to write programs, the programs that it writes are free from logical errors. As shown in Section 6, PROW can also be used to aid robots, to design circuits as well as to write programs in languages other than LISP.

The authors have tried hard to make this paper self-contained. However, the readers who want to thoroughly understand "PROW" should be familiar with LISP (3,10) and automatic theorem proving (13,6,8).

It should be noted that the function of PROW is quite different from that of a compiler because the user of a compiler, such as a FORTRAN programmer, has to construct the algorithm himself, while the user of PROW has only to supply the specifications of his program and PROW is able to decide the algorithm for him.

The proof of the correctness of the algorithm used in PROW is omitted in this paper because of the limitation of space. It will be included in R. Waldinger's Ph.D. thesis later.

Section 2: The Resolution Principle and Program

The theorem prover of PROW uses Robinson's resolution principle (5,6,8) as its inference rule. It will be shown later that the ordinary resolution principle is not appropriate and adequate for program writing. This leads to the notion of "primitive resolution" which will be discussed in detail in Section 3.

Example 1: The characteristic function of the predicate "Atom" in LISP.

Suppose we want to write a program whose output is 1 if the input is an atom and 0 otherwise.

The axiom we need is

$$\text{Equal}(X,X) \qquad (1)$$

The theorem to be proven is

$$(X)(EY)($$

$$(\text{Atom}(X) \;\&\; \text{Equal}(Y,1))$$

$$\text{V } (\sim\text{Atom}(X) \;\&\; \text{Equal}(Y,0)) \;) \qquad (2)$$

The negation of the theorem with quantifiers removed, (see (5)), consists of the following clauses:

$$\sim\text{Atom}(a) \text{ V } \sim \text{Equal}(Y,1) \qquad (3)$$

$$\text{Atom}(a) \text{ V } \sim \text{Equal}(Y,0) \qquad (4)$$

The proof of the theorem is

(1) & (3) $\sim$Atom(a) (5)

(1) & (4) Atom(a) (6)

(5) & (6) NIL (7)

The proof can be represented by a tree, where every node represents a clause in the proof, and under every arc, we write down the following information:

(a) The substitution of the variables, if there is any substitution made.

(b) If clause N is derived from clauses N1 and N2, then on the arc from N to N1, write down the negation of the literal deleted from clause N1, with the appropriate substitutions made.

The proof of Example 1 can now be represented by the tree in Fig. 2-1.

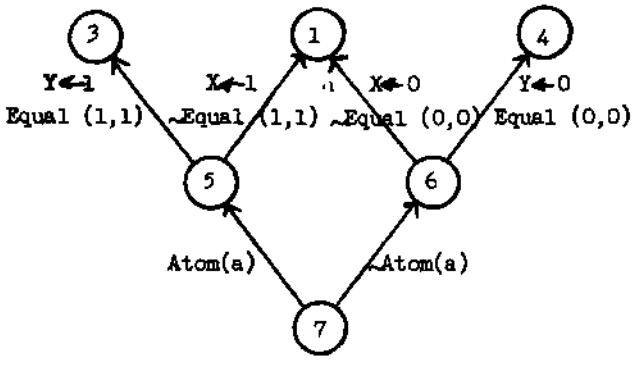

Fig. 2-1

The tree in Fig. 2-1 can be considered as the flowchart of the program. Nodes 3,1 and 4 are halt or terminal points and node 7 is the starting point. The predicates represent testing points for traversing the arcs. For instance, the arc from (5) to (3) is labelled with the test "Equal(1,1)". This means that if control is at node (5) and the predicate Equal(1,1) is true, then control passes to node (3). Since the arc is also labelled with the substitution y←1, the variable y is set to 1.

In Fig. 2-1, the arcs leading to (1) are of special interest because ~Equal(1,1) and ~Equal(0,0) can never be true. Therefore, control will never be passed to node (1). In fact, it can be shown that if a node represents an axiom or a clause which is always true, then control will never be passed to that point. We can thus safely delete node (1) from the tree and Fig.2-1 is simplified to Fig.2-2.

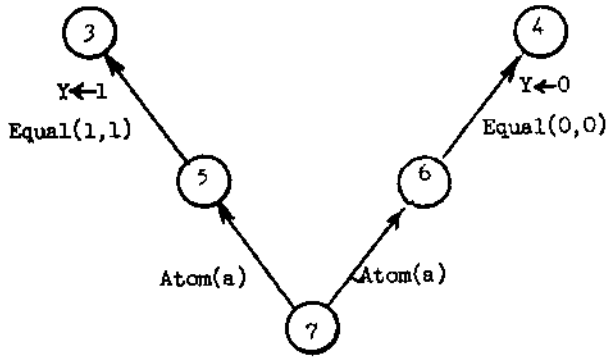

Fig. 2-2

Since there is no branching out of nodes (5) and (6), no testing is necessary in those two nodes. We can therefore further simplify Fig.2-2 to Fig.2-3.

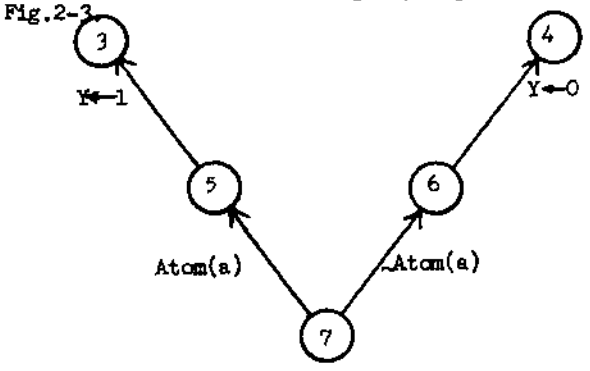

Fig. 2-3

The LISP program produced by PROW is the following:

```
(LAMBDA (A)

(PROG  (Y)

(COND  ((ATOM A) (GO A1)))

(SETQ Y 0)

(RETURN Y)

 A1 (SETQ Y 1)

(RETURN Y)  ))
```

Example 2: The "Member" function.

The "Member" function of LISP is a widely used function. The input has two arguments: X and L, where X is any S-expression and L is a list. The output will be T if $X \in L$ and NIL if otherwise.

The axioms expressing the definition of "Member" are

$$X \notin L \quad V \ \text{Equal(Member(X,L),T)} \qquad (8)$$

$$X \in L \quad V \ \text{Equal(Member(X,L),NIL)} \qquad (9)$$

The theorem to be proven is

$$(x)(L)(EY)\ \text{Equal(Member(X,L),Y)} \qquad (10)$$

The Negation of the theorem gives

$$\sim\text{Equal(Member(a,b),Y)} \qquad (11)$$

(8) & (11) $\quad X \notin L \qquad (12)$

(9) & (11) $\quad X \in L \qquad (13)$

(12) & (13) $\quad$ NIL $\qquad (14)$

The tree representing the proof of Example 2 looks like Fig.2-4.

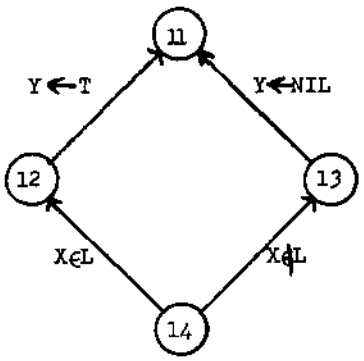

Fig. 2-4

The program represented ty the tree in Fig.2-4 will necessarily involve the testing of $X \in L$, which is not defined or primitive in LISP. Therefore, the program is meaningless. Proofs such as the one described above which yield non-executable programs must be forbidden. This is achieved as follows: the user of PROW may declare which symbols are primitive. Only primitive symbols will appear in the program.This is made possible Tpy using a special inference rule, called "primitive resolution." The proof that is obtained by "primitive resolution" is called a "primitive proof." The program that is obtained by processing a primitive proof contains only primitive symbols. This will be discussed in the next section.

Section 3: The Primitive Proof and the Post-Proof PrPtteBSPr

In this section, we are going to define a new inference rule, called the "primitive resolution." The proof that is obtained 'by the application of the primitive resolution is called the "primitive proof." For readers who are interested only in the performance of PROW, this section can be skipped.

First of all, the user of PROW has to declare the following:

(a) Primitive symbols: The symbols which are allowed to appear in the program.

(b) Insipid clauses: The clauses that the user claims to be always true. (All the axioms are necessarily insipid, but whether a special hypothesis Is insipid or not depends on the user.)

(c) Output variables: Suppose the theorem concerning the program has the following quantifier:

$$(X_1)(X_2) \ldots (X_m)(EY_1)(EY_2) \ldots (EY_n)$$

Negation of the theorem will make all the variable X 's appear in the clauses as constants (4). They are called the input constants. All the Y_i's will appear as variables in the clauses and the user has to declare some of the i_1's as output variables.

We also need the following definitions:

Def.: Halt clause: the set of halt clauses is the set of all the clauses that the user provides as input.

Def.: Vital clause: the definition of a vital clause is inductive.

(a) A halt clause that is not declared insipid is a vital clause.

(b) In a proof, a clause B that is a resolvent of clauses C and D is vital iff one or both of C and D is vital.

Def.: Hot variables: the hot variables are also defined inductively.

(a) Insipid clauses have no hot variables.
(b) The hot variables of a vital halt clause are those output variables (if any), appearing in that clause.
(c) Let B be a resolvent of clauses C and D. Say $B=(C-L)\sigma_c \cup (D-M)\sigma_d$

(1) If W is hot in C, then the variables which occur in both $W\sigma_c$ and B are hot in B. If W is hot in D, then the variables which occur in $W\sigma_d$ and B are hot in B.

(2) If C and D are both vital, then the variables which occur in both $L\sigma_c$ and B are also hot in B.

Example:

Let $C = \sim P(X) \vee Q(Y)$
and $D = P(X) \vee R(X)$.

Suppose that C and D are both vital and Y is the only hot variable. The resolvent of C and D is

$B = Q(Y) \vee R(X)$.

B is vital and its hot variables are Y and X. The variable Y is hot because it occurs in a term of the form $W\sigma_c$ (W=Y and σ_c =the identity substitution). (Condition (1)). The variable X is hot because it occurs in a term of the form $L\sigma_c$ where L=P(X), σ_c = the identity substitution and because C and D are both vital. (Condition (2)).

Def.: A proof P is said to be "primitive" if it obeys the following restrictions:

(a) Let clause B be a resolvent of clauses C and D, say $B=(C-L)\sigma_c \cup (D-M)\sigma_d$

Then if W is hot in C, all the constant and function symbols occurring in $W\sigma_c$ must be primitive. If W is hot in D, then all the constant and function symbols occurring in $W\sigma_d$ must be primitive.

(b) If C and D are both vital, then all the constant, function and predicate symbols occurring in $L\sigma_c$ must be primitive.

Example: The proof of the first example in Section 2 is primitive since Atom, a, 0, and 1 are all primitive. The predicate "Atom" is a standard LISP predicate, the constant a is the input constant and 0 and 1 are numerals which we can use in LISP computations.

If, for some reason, 1 were not primitive, the deduction of clause 5 would be illegal in primitive resolution since the variable Y is hot in clause 3 and the resolution involves substituting 1 for Y by condition (a). On the other hand, if "Atom" were not primitive, then the deduction of clause 7 would be illegal by condition (b).

After the theorem is proved, the proof which is a sequence of clauses is fed into a post-proof processor. Before describing the processor, we first have to define the following:

Def.: Substitution: A substitution is a finite set

$\{T_1/V_1, T_2/V_2, \ldots, T_k/V_k\}$

of substitution components T_i/V_i, where V_i is a variable and T_i is a term different from V_i, and no two components in the set have the same variable after the stroke symbol.

Def.: Operation $P(\sigma,X)$

Let σ be a substitution and let X be a set of variables.

(1) If σ is the empty substitution, then $P(\sigma,X)$ is empty.

(2) If $\sigma = \sigma_1 \cup \tau$ where τ is a substitution component, and the variable of τ does not belong to X, then $P(\sigma,X)=P(\sigma_1,X)$.

(3) Otherwise, suppose $\tau = T_i/V_i$, then $P(\sigma,X) = (\text{SETQ } V_i\ T_i)P(\sigma_1,X)$.

For example,

P ({A/X, B/Y, (F A)/Z}, {X,Z})

= ((SETQ Z (F A)) (SETQ X A))

The post-proof processor would first single out vital clauses of the proof and with each vital clause B, the processor associates a program as follows:

1. Suppose B is a halt clause. Suppose $Y_1, Y_2, \ldots, Y_n$ is the complete list of output variables, including those not appearing in B. The program associated with B is then

(RETURN (LIST Y_1 Y_2 ... Y_n))

2. Suppose B is a resolvent of clauses C and D where C at least is vital, say $B = (C-L)\sigma_c \cup (D-M)\sigma_d$. Let H_C be the set of hot variables of C, and let H_D be the set of hot variables of D. Let ProgC and ProgD be the programs associated with clauses C and D respectively.

(a) Suppose C and D are both vital. Let GC be a label which does not appear in the program associated with any other clause in the proof. Then the program ProgB associated with B is defined by

ProgB

$= \{[M \sigma_D$ -go [GC]];

$P(\sigma_D, H_D)$;

ProgD

GC: $P(G_C, H_C)$

ProgC}

(a) Suppose C is vital, but D is not. Then

ProgB = $P(\sigma_C, H_C)$ ProgC.

Let $C_1, C_2, \ldots, C_k$ be the list of input constants.

Let Progn be the program associated with the empty clause in the proof. Let $X_1, X_2, \ldots X_j$ be the complete list of variables in Progn. Then the final program that the machine produces as output is

lambda $[C_1, C_2, \ldots, C_k]$[prog $[X_1, \ldots, X_j]$ Progn]

Section 4: The Induction Axiom and Looping in a Program

The algorithm we gave in the last section will not produce any program with recursive property, i.e., the programs will not be able to call themselves. In PROW, the recursiveness of a program is provided by induction schema. By using an induction scheme, we mean that instead of proving the main theorem, it suffices to prove some sub-theorems. Every proof of a sub-theorem produces a sub-program. PROW also has the ability to combine all the sub-programs together to form the main program. We shall illustrate these procedures by an example first and give the exact algorithm at the end of this section.

Example: We want to write the program that computes the "factorial" function which is defined inductively as follows:

$$f(0) = 1 \tag{1}$$

$$f(n) = n \cdot f(n-1) \tag{2}$$

Let predicate Factrel(X,Y) denote f(X) = Y. Then the axioms concerning the definition of "factorial" are:

$$\text{Factrel}(0,1) \tag{3}$$

$$\neg\text{Factrel}(X,Y) \vee \text{Factrel}(\text{Add1}(X), \text{Times}(\text{Add1}(X),Y)) \tag{4}$$

where Add1(X) and Times (X,Y) are both LISP functions. (Add1(X)=X+1 and Times(X,Y)=X·Y). The theorem to prove in this case is

$$(X)(EY)\text{Factrel}(X,Y) \tag{5}$$

The negation of the theorem gives the following clause:

$$\neg\text{Factrel}(a,Y) \tag{6}$$

where a is the input constant and Y is the output variable. "Factrel" is not considered as primitive, of course.

To prove this theorem, we need a famous induction axiom for natural numbers. Let R(X,Y) mean f(X)=Y where f(X) is any function of natural numbers. Then the induction axiom can be expressed as follows:

$$\begin{aligned}&(\quad (EY_1)R(0,Y_1) \\ \& \quad &(X_2)(\quad (EY_2)R(X_2,Y_2) \rightarrow (EY_3)R(\text{Add1}(X_2),Y_3))) \\ \rightarrow &(X)(EY)R(X,Y)\end{aligned} \tag{7}$$

Formula (7) suggests that, instead of proving

$$(X)(EY)\text{Factrel}(X,Y)$$

directly, we prove the following two sub-theorems:

$$(EY)\text{Factrel}(0,Y) \tag{8}$$

and

$$(X)((EY_1)\text{Factrel}(X,Y_1) \rightarrow (EY_2)\text{Factrel}(\text{Add1}(X), Y_2)) \tag{9}$$

The negation of (8) gives

$$\neg\text{Factrel}(0,Y) \tag{10}$$

The negation of (9) gives

$$\text{Factrel}(a,b) \tag{11}$$

$$\& \quad \neg\text{Factrel}(\text{Add1}(a),Y) \tag{12}$$

Let the programs associated with (8) and (9) be Prog1 and Prog2 respectively. Prog1 has no input parameter while Prog2 has a and b as input parameters.

Besides the induction axiom, PROW also stores in its memory the information on how to connect Prog1 and Prog2 together to form the program computing f(X). For this case, the function f(X) can be expressed in terms of Prog1 and Prog2 in the following form:

f(X)

= { equal [X;0] → Prog1;

T → Prog2 {a;b} {Subl [X]; f [Subl[x]]}]

where Subl [x] = x-1.

Prog1 and Prog2 can be found by using the algorithm introduced in Section 3. The reader can easily convince himself that:

```
Prog A
=    (LAMBDA ( )
     (PROG (Y)
     (SETQ Y 1)
     (RETURN Y) ))
ProgB
=    (LAMBDA (B C)
     (PROG (Y)
     (SETQ Y (TIMES (ADD1 B) C))
     (RETURN Y) ))
```

The program that computes "factorial" is thus

```
Factorial
=    (LAMBDA (X)
     (COND ((EQUAL X 0)
        (LAMBDA ( )
        (PROG (Y)
        (SETQ Y 1)
        (RETURN Y)))))
     (T ((LAMBDA (B C)
        (PROG (Y)
        (SETQ Y (TIMES (ADD1 B) C))
        (RETURN Y)))
          (SUB1 X) (FACTORIAL (SUB1 X))))  ))
```

In the LISP system, we may also introduce induction axioms. For example, the following axiom is a very useful one:

$$\begin{aligned}&(\ (EY_1)R(NIL,Y_1)\\ \&\quad &(X_2)(Z_2)(\ (EY_2)R(X_2,Y_2)\\ &\qquad \rightarrow (EY_3)R(cons(Z_2,X_2),Y_3)))\\ &\rightarrow (X)(EY)R(X,Y)\end{aligned} \tag{14}$$

Again, R(X,Y) means f(X)=Y.

Roughly speaking, this means that if

(1)NIL has property P, and (2) cons(y,x) has property P, whenever X has property P, then every S-expression that is expressible in "list" notation has property P.

Again, the induction axiom will suggest two sub-theorems to prove:

$$(EY)R(NIL,Y) \tag{15}$$

$$\&\ (X)(Z)(\ (EY_1)R(X,Y_1)$$

$$\rightarrow (EY_2)R(cons(Z,X),Y_2) \tag{16}$$

The negation of (15) gives

$$\sim R(NIL,Y) \tag{17}$$

and the negation of (16) gives

$$R(a,c) \tag{18}$$

$$\sim R(cons(b,a),Y) \tag{19}$$

The program associated with (17), Prog1, has no input arguments while the program associated with (18) & (19), Prog2, has a,b & c as input arguments. The function that computes f(x) in terms of Prog1 and Prog2 takes the following form:

```
f(x)
=  [ null [x]  → Prog1;
     T → Prog2[a,b,c] [cdr[x];car[x];f[cdr[x]]]]
```

If there are two input variables, then the induction axiom may take the following form:

$$\begin{aligned}&(X_1)(EZ_1)R(X_1,NIL,Z_1)\\ \&\quad &(V_2)(Y_2)\ (\ (X_2)(EZ_2)R(X_2,Y_2,Z_2)\\ &\rightarrow (X_3)(EZ_3)R(X_3,cons(V_2,Y_2),Z_3))\\ &\rightarrow (X)(Y)(EZ)R(X,Y,Z)\end{aligned} \tag{20}$$

Formula (20) is a very useful axiom. It can be used to write the "Member" function in LISP. We shall devote the whole next section to show how this is done.

Section 5: The "Member" function of LISP

Due to the limited capability of the theorem prover of PROW, PROW cannot prove the theorems involved in the "Member" function of LISP. The following example is a hand-simulated one.

The "Member" function is defined as follows:

Input: X is an S-expression.
L is a list.

Output: Y = T if $X \in L$
Y = NIL if otherwise.

Let P(X,L) denote $X \in L$, Memrel(X,L,Y) denote Member(X,L) = Y and Equal(X,Y) denote X = Y.

The axioms concerning the definition of

"member"; the property of P(X,L) as well as the relationship between P(X,L) and Equal (X,Y) are collected as follows:

$\sim P(X,L) \vee Memrel(X,L,T)$ (1)

$P(X,L) \vee Memrel(X,L,NIL)$ (2)

$\sim Equal(X,Y) \vee P(X,cons(Y,L))$ (3)

$\sim P(X,L) \vee P(X,cons(Y,L))$ (4)

$Equal(X,Y) \vee P(X,L) \vee \sim P(X,cons(Y,L))$ (5)

$\sim Equal(X,Y) \vee \sim Equal(Y,Z) \vee Equal(X,Z)$ (6)

$\sim Equal(X,Y) \vee Equal(Y,X)$ (7)

$\sim Equal(X,Y) \vee \sim Equal\ (Z,Y) \vee Equal(X,Z)$ (8)

$\sim P(X,NIL)$ (9)

$\sim Memrel(X,L,U) \vee \sim Equal(U,V) \vee Memrel(X,L,V)$ (10)

$\sim Memrel(X,L,U) \vee \sim Memrel(X,L,V) \vee Equal\ (U,V)$ (11)

$\sim Equal(Member(X,L),U) \vee Memrel(X,L,U)$ (12)

$Equal(X,X)$ (13)

The main theorem to prove is

$$(X)(L)(EY)Memrel(X,L,Y)$$

The negation of this theorem is

$$\sim Memrel(a,b,Y) \quad (14)$$

The Induction Axiom is

$$(\ (X_1)(EY_1)R(X_1,NIL,Y_1)$$

$$\&\ (L_1)(\ (X_2)(EY_2)R(X_2,L_1,Y_2)$$

$$\rightarrow (X_3)(X_4)(EY_3)R(X_3,cons(X_4,L_1),Y_3)))$$

$$\rightarrow (X)(L)(EY)R(X,L,Y) \quad (15)$$

From (15), we have to prove the following two theorems:

First theorem:

$$(X)(EY)Memrel(X,NIL,Y) \quad (16)$$

The negation of this theorem is

$$\sim Memrel(a_1,NIL,Y) \quad (17)$$

Second theorem:

$$(L)(\ (X_1)(EY_1)Memrel(X_1,L,Y_1)$$

$$\rightarrow\ (X_2)(X_3)(EY_2)Memrel(X_2,cons(X_3,L),Y_2)) \quad (18)$$

The above theorem is equivalent to:

$$(L)(\ (EX_1)(Y_1)\sim Memrel(X_1,L,Y_1)$$

$$\vee\ (X_2)(X_3)(EY_2)Memrel(X_2,cons(X_3,L),Y_2)) \quad (19)$$

The negation of this theorem gives

$$Memrel\ (X,a_2,f(X)) \quad (20)$$

$$\sim Memrel\ (b_2,cons(c_2,a_2),Y) \quad (21)$$

The following symbols are declared "primitive."

(A) Constants: NIL, T, a_1,a_2, b_2, c_2.

(B) Functions: f(X), Member(X,L),cons(X,Y)

(C) Predicates: Equal(X,Y)

The vital clauses are (17) & (21). In each of them, the output variable Y is a hot variable. Let the programs associated with the first theorem and the second theorem be Prog1 and Prog2 respectively. The function that combines Prog1 and Prog2 is

```
    f(x,l)
= [[null [l] → Prog1;
   T → Prog2[a2,b2,c2] [cdr[l],x,car[l]]]
```

To help the readers understand the resolution principle which is used in PROW to prove theorems, we write (C_1,l_1,C_2,l_2) in front of clause C in case that clause C is a resolvent of clauses C_1 and C_2 by deleting the l_1th literal of C_1 and the l_2th literal of C_2. If clause C is a factor of clause C_1 by matching the l_1th and the l_2th literals of C_1, then we write (C_1,L_1,C_1,L_2) in front of clause C.

The proof of the second theorem:

(1,1,3,2) $\sim Equal(X,Y) \vee Memrel(X,cons(Y,L),T)$ (22)

(1,1,4,2) $\sim P(X,L) \vee Memrel(X,cons(Y,L),T)$ (23)

(1,2,11,2) $\sim P(X,L) \vee \sim Memrel(X,L,U) \vee Equal(U,T)$ (24)

(23,2,11,2) $\sim P(X,L) \vee \sim Memrel(X,cons(Y,L),U) \vee Equal(U,T)$ (25)

(24,3,8,1) $\sim P(X,L) \vee \sim Memrel(X,L,U) \vee \sim Equal(Z,T) \vee Equal(U,Z)$ (26)

(25,3,26,3) $\sim P(X_1,L_1) \vee \sim P(X_2,L_2) \vee \sim Memrel(X_1,cons(Y_1,L_1),U) \vee \sim Memrel(X_2,L_2,V) \vee Equal(V,U)$ (27)

(2,1,5,3) $Equal(X,Y) \vee P(X,L) \vee Memrel(X,cons(Y,L),NIL)$ (28)

(28,3,11,2) $Equal(X,Y) \vee P(X,L) \vee \sim Memrel(X,cons(Y,L),V) \vee Equal(V,NIL)$ (29)

(2,2,11,1) P(X,L) V ~Memrel(X,L,V) V Equal(V,NIL) (30)

(29,4,8,1) Equal(X,Y) V P(X,L) V ~Memrel(X,cons(Y,L),V) V ~Equal(Z,NIL) V Equal(V,Z) (31)

(31,4,30,3) $P(X_1,L_1)$ V ~Memrel(X_1,L_1,U) V Equal(X_2,Y) V $P(X_2,L_2)$ V ~Memrel(X_2,cons(Y,L_2),V) V Equal(V,U) (32)

(32,1,32,4) P(X,L) V Equal(X,Y) V ~Memrel(X,L,U) V ~Memrel(X,cons(Y,L),V) V Equal(V,U) (33)

(27,1,27,2) ~P(X,L) V ~Memrel(X,cons(Y,L),U) V ~Memrel(X,L,V) V Equal(V,U) (34)

(33,1,34,1) $Equal(X,Y_1)$ V ~Memrel(X,L,U_1) V ~Memrel(X,cons(Y_1,L),V_1) V $Equal(V_1,U_1)$ V ~Memrel(X,cons(Y_2,L),U_2) V ~Memrel(X,L,V_2) V $Equal(V_2,U_2)$ (35)

(35,2,35,6) $Equal(X,Y_1)$ V ~Memrel(X,L,U_1) V ~Memrel(X,cons(Y_1,L),V_1) V $Equal(V_1,U_1)$ V ~Memrel(X,cons(Y_2,L),U_2) V $Equal(U_1,U_2)$ (36)

(36,3,36,5) Equal(X,Y) V ~Memrel(X,L,U) V ~Memrel(X,cons(Y,L),V) V Equal(V,U) V Equal(U,V) (37)

(37,5,7,1) Equal(X,Y) V ~Memrel(X,L,U) V ~Memrel(X,cons(Y,L),V) V Equal(V,U) (38)

(38,3,12,2) Equal(X,Y) V ~Memrel(X,L,U) V Equal(V,U) V ~Equal(Member(X,cons(Y,L)),V) (39)

(39,4,13,1) Equal(X,Y) V ~Memrel(X,L,U) V Equal(Member(X,cons(Y,L)),U) (40)

(40,3,12,1) Equal(X,Y) V ~Memrel(X,L,U) V Memrel(X,cons(Y,L),U) (41)

(41,2,20,1) Equal(X,Y) V Memrel(X,cons(Y,a_2),f(X)) (42)

*(42,2,21,1) $Equal(b_2,c_2)$ (43)

*(22,2,21,1) ~$Equal(b_2,c_2)$ (44)

*(44,1,43,1) NIL (45)

In the above proof, those clauses that have "*" signs attached are vital clauses. The programs corresponding to these vital clauses are the following:

```
Prog43   (SETQ Y (MEMBER B A))
         (RETURN Y)

Prog44   (SETQ Y T)
         (RETURN Y)

Prog45   (COND((EQUAL B C) (GO A1)) )
         (SETQ Y (MEMBER B A))
         (RETURN Y)
   A1    (SETQ Y T)
         (RETURN Y)
```

So, Prog2 corresponding to the second theorem is

```
Prog2 =  (LAMBDA (A B C)
         (PROG (Y)
         (COND ((EQUAL B C)(GO A1)) )
         (SETQ Y (MEMBER B A))
         (RETURN Y)
   A1    (SETQ Y T)
         (RETURN Y)) )
```

It now remains to find the program corresponding to the first theorem. The proof is much easier as compared with that of the second theorem.

Proof of the first theorem:

* (17,1,2,2) $P(a_1$,NIL) (46)

* (46,1,9,1) NIL (47)

The programs associated with clauses (46) and (47) are

```
Prog46   (SETQ Y NIL)
         (RETURN Y)

Prog47   (SETQ Y NIL)
         (RETURN Y)
```

The program associated with the first theorem is

```
Prog1 = (LAMBDA ( )
        (PROG (Y)
        (SETQ Y NIL)
        (RETURN Y) ))
```

Let us restate the function combining these two programs together:

```
        f(x,l)
    = [ null [l]  → Prog1;
      T  →Prog2[a2,b2,c2][cdr [l],x,car [l]]]
      Member
   =(LAMBDA (X L)
    (COND ((NULL L) (LAMBDA ( )
                    (PROG (Y)
                    (SETQ Y NIL)
                    (RETURN Y) )))
          (T  ((LAMBDA (A B C)
                    (PROG (Y)
                    (COND ((EQUAL B C) (GO A1)))
                    (SETQ Y (MEMBER B A))
                    (RETURN Y)
               A1   (SETQ Y T)
                    (RETURN Y)))
                     (CDR L) X (CAR L)) )))
```

Section 6: <u>The Application of PROW to Deductive Question-Answering</u>

Since the output of PROW can be considered as a sequence of instructions, it can also be used as a question-answering program (1,8). In this section, we shall show four examples of how this can be done.

Example 1: The first monkey-banana problem.

The problem can be briefly stated as follows: A banana is suspended from the ceiling of a room. A monkey is on the floor, but is too short to reach the banana. There is a chair in the room, and the monkey can walk freely in the room, carrying the chair together with him. Furthermore, the monkey can climb the chair.

This problem has been solved by the programs of Slagle (8) and Green (1).

Let P(XM, ZM, XC, S) represent the sentence, "State S is attainable in which the monkey is at position XM on the floor, at height ZM while the chair is at the horizontal position XC on the floor."

We also define the following functions:

(1) Walk(X,S): the state achieved if the monkey is in state S and then walks to position X.

(2) Carry(X,S): the state achieved if the monkey is in state S and walks to position X, carrying the chair with him.

(3) Climb(X,S): the state achieved if the monkey is in state S and climbs the chair whose horizontal position is X.

The problem can be axiomatized as below:

$$\sim P(X,O,Y,Z) \vee P(W,O,Y,Walk(W,Z)) \quad (1)$$

$$\sim P(X,O,X,Y) \vee P(W,O,W,Carry(W,Y)) \quad (2)$$

$$\sim P(X,O,X,Y) \vee P(X,W,X,\ Climb\ (W,Y)) \quad (3)$$

The theorem is

$$(X_1)(Y_1)(Z_1)(S_1)(X_2)(Y_2)(EZ_2)(ES_2)$$

$$(P(X_1,Y_1,Z_1,S_1) \rightarrow P(X_2,Y_2,Z_2,S_2)) \quad (4)$$

The negation of the theorem gives

$$P(A,B,C,D) \quad (5)$$

$$\sim P(E,H,X,Y) \quad (6)$$

All the constants as well as the function symbols are primitive. The output variable is Y in (6) which is the final state.

PROW generates a program as below:

```
(LAMBDA (A B C D E H)
(PROG (Z3 Z2 Z)
(SETQ Z3 D)
(SETQ Z2 C)
(SETQ Z2 (WALK Z2 Z3))
(SETQ Z2 (CARRY E Z2))
(SETQ Z (CLIMB H Z2))
(RETURN (LIST Z))  ))
```

The reader can see that the program describes the solution of the problem as the following steps:

(1) The monkey walks from where originally he stands to the chair.

(2) The monkey walks from the position of the chair to the position of the banana, carrying the chair with him.

(3) The monkey climbs the chair.

In the next example, we shall show a very important feature of PRCW. That is, if used as a question-answering program, it can answer questions whose answers involve "conditional expressions." For example, it may say, "If the door is locked, use the window." This is a great improvement of the present question-answering programs. Also, in this example, the reader can see the importance of the "primitive resolution."

Example 2: There are two boxes, one and only one of which has a banana in it. The monkey can walk freely and he does not know which box contains the banana. But if the monkey is where the box is, he can look into the box and see whether the box contains the banana or not. What should the monkey do to get the banana?

Let P(X,Y,S) mean that "the state S is attainable in which the monkey is at X and the banana is at Y."

Let Find(S) mean that "S is the state in which the monkey can find the banana."

Axioms:

$$\sim P(X,X,S) \vee Find(S) \qquad (1)$$

$$\sim P(X,Y,S) \vee P(W,Y,Walk(W,S)) \qquad (2)$$

The theorem is

$$(X_1)(Y_1)(S_1)(Y_2)$$

$$(P(X_1,Y_1,S_1) \vee P(X_1,Y_2,S_1)) \rightarrow (\exists S_2)Find(S_2) \qquad (3)$$

This theorem says that no matter where the banana is, the monkey can find it.

The negation of (3) is

$$P(A,B,C) \vee P(A,D,C) \qquad (4)$$

$$\sim Find(S) \qquad (5)$$

The primitive symbols are A,B,C,D,Find, and Walk. The vital clause is (5) and the hot variable is S.

Proof:

(1,2,5,1)	$\sim P(Z_1,Z_1,Z_2)$	(6)
(2,2,6,1)	$\sim P(Z_1,Z_2,Z_3)$	(7)
(7,1,4,1)	$P(A,D,C)$	(8)
(8,1,2,1)	$P(Z_1,D,Walk(Z_1,C))$	(9)
(9,1,1,1)	$Find(Walk(D,C))$	(10)
(10,1,5,1)	NIL	(11)

Applying the algorithm given in Section 2 again, we obtain the following program:

```
   (LAMBDA (A B C D)
   (PROG (Z3, Z2, Z)
   (COND ((NOT (FIND (WALK D C)))(GO A1)) )
   (SETQ Z (WALK D C ))
   (RETURN Z)
A1 (SETQ Z3 C)
   (SETQ Z2 B)
   (SETQ Z2 (WALK Z2 Z3))
   (SETQ Z Z2)
   (RETURN Z) ))
```

The program can be interpreted as:

1. Go to box B.
2. If you can find the banana in box B, then go back to the starting point and go to box B.
3. Otherwise, go to box A.

The unfortunate thing is that, as one can see, the monkey has to go back to the starting point. This is simply because PROW is designed for a program writer. Therefore it ignores the side-effect of a testing.

If we allow P(X,Y,S) to be primitive, we would get the following solution:

```
   (LAMBDA (A B C D)
   (PROG (Z3 Z2 Z)
   (COND ((NOT (P A D C)) (GO A1)))
   (SETQ Z3 C)
   (SETQ Z2 D)
   (SETQ Z2 (WALK Z2 Z3))
   (SETQ Z Z2)
   (RETURN Z)
A1 (SETQ Z3 C)
   (SETQ Z2 B)
   (SETQ Z2 (WALK Z2 Z3))
   (SETQ Z Z2)
   (RETURN Z) ))
```

This program involves a testing of the predicate P(A,B,C) which means that the monkey should decide whether the banana is at box B or not. This is, of course, not desirable.

In the third example, PROW is asked to design computer circuits.

Example 3: We have to construct an "Or" gate out of "And" and "Not" gates.

Let I(X) mean X has value T.

Let And(X,Y) = X & Y

Or(X,Y) = X V Y

and Not(X) = ~ X

The axioms for this problem are

~ I(And(X,Y)) V I(X) (1)

~ I(And(X,Y)) V I(Y) (2)

~ I(X) V ~I(Y) V I(And(X,Y)) (3)

I(Not(X)) V I(X) (4)

~ I(Not(X)) V ~I(X) (5)

I(Or(X,Y)) V ~I(X) (6)

I(Or(X,Y)) V ~I(Y) (7)

~ I(Or(X,Y)) V I(X) V I(Y) (8)

~ I(X) V ~I(Y) V Equal(X,Y) (9)

I(X) V I(Y) V Equal(X,Y) (10)

~ I(X) V I(Y) V ~Equal(X,Y) (11)

I(X) V ~I(Y) V ~Equal(X,Y) (12)

The functions "And" and "Not" are declared primitive and "Or" is not. Both predicates "Equal" and "I" are not primitive. Otherwise, we might be vulnerable to conditional expressions appearing in our circuits, as if they were LISP programs.

The theorem to be proved is

(X)(Y)(EZ) Equal(Or(X,Y),Z) (13)

The proof is omitted. By tracing the hot variable Z, we obtain

Z ⟵ Not(And(Not(X),Not(Y)))

which is the circuit we want.

In the present status, PROW can only write programs in LISP. Actually, the idea of PROW is general enough to write programs in any language. In the next example, we shall show how PROW can handle problems in machine language.

Example 4: Let us imagine a machine with three registers A,B and C and one accumulator. We would like to write a program to reverse the contents of registers A and B.

Let P(U,X,Y,Z,S) mean that "State S in which the accumulator contains U and registers A, B and C contain X, Y, and Z respectively is attainable."

Let loada(s), loadb(s) and loadc(s) be the states after the accumulator is loaded with the contents of registers A, B and C in state S, respectively.

Let Storea(s), Storeb(s) and Storec(s) be the states after the content of the accumulator is stored into registers A, B and C in state S respectively.

The entire machine can be axiomatized as follows:

~ P(U,X,Y,Z,S) V P(U,U,Y,Z,loada(s)) (1)

~ P(U,X,Y,Z,S) V P(U,X,U,Z,loadb(s)) (2)

~ P(U,X,Y,Z,S) V P(U,X,Y,U,loadc(s)) (3)

~ P(U,X,Y,Z,S) V P(U,U,Y,Z,Storea(s)) (4)

~ P(U,X,Y,Z,S) V P(U,X,U,Z,Storeb(s)) (5)

~ P(U,X,Y,Z,S) V P(U,X,Y,U,Storec(s)) (6)

The theorem is

$(U_1)(X)(Y)(Z_1)(S_1)(EU_2)(EZ_2)(ES_2)$

$(P(U_1,X,Y,Z_1,S_1) \supset P(U_2,Y,X,Z_2,S_2))$ (7)

The negation of the theorem is

P(Acc,A,B,C,D) (8)

~P(U,B,A,Z,S) (9)

The program that PROW produces is the following:

```
(LAMBDA (D)
(PROG (Z3 Z)
(SETQ Z3 D)
(SETQ Z3 (LOADA Z3))
(SETQ Z3 (STOREC Z3))
(SETQ Z3 (LOADB Z3))
(SETQ Z3 (STOREA Z3))
(SETQ Z3 (LOADC Z3))
(SETQ Z (STOREB Z3))
(RETURN Z) ))
```

This program, although written in LISP, represents a machine language program. The sequence

of instructions is: Load the accumulator with register A. Store it into register C. Load the accumulator with register B. Store it into register A. Load the accumulator with register C and store it into register B. The contents of registers A and B are now reversed.

Section 7: Conclusions

The ultimate goal of developing PROW is to have a system such that the programmers are not required to know particular computer languages thoroughly and do not have to construct the algorithms. If this goal is achieved, programming would be more delightful and less time consuming. As yet, PROW is only a step towards this goal.

At present, PROW only understands the first order predicate calculus. It is therefore sometimes more difficult to specify a program than writing the program directly. PROW would be much more improved if some easier-to-use language, such as English, replaces predicate calculus as its input language.

As one can see, the efficiency of PROW depends heavily upon the efficiency of the theorem prover that PROW uses. It is absolutely necessary to improve the efficiency of the theorem prover.

Acknowledgements

The authors wish to express their gratitude to Professor H. A. Simon of Carnegie-Mellon University, Dr. J. R. Slagle and Dr. C. L. Chang of the National Institutes of Health for their advice, guidance and constructive suggestions.

References

1. Green, C. and Raphael, G.: The Use of Theorem-Proving Techniques in Question-Answering Systems, Proceedings of the 1968 ACM National Conference, 169-181.

2. Hilbert, A. and Ackerman, W.: Principles of Mathematical Logic, Chelsea Publishing Company, N.Y. (1950).

3. McCarthy, J.: LISP 1.5 Programmer's Manual, the M.I.T. Press, Cambridge, Mass. (1962).

4. Quine, W.: A Proof Procedure for Quantification Theory, Journal of Symbolic Logic, Vol. 20, No.2, (June,1955), 141-149.

5. Robinson, J.: A Machine Oriented Logic Based on the Resolution Principle, JACM, Vol. 7, (Jan.,1965), 23-41.

6. Robinson, J.: A Review of Automatic Theorem Proving, Annual Symposia in Applied Mathematics, Vol.XIX,(1966).

7. Simon, H.A.: Experiments with a Heuristic Compiler, JACM, (Oct., 1963), 482-506.

8. Slagle, J.: Experiments with a Deductive Question-Answering Program, Comm.ACM, Vol.8, No.12,(Dec.,1965) 792-798.

9. Slagle, J.: Automatic Theorem Proving with Renameable and Semantic Resolution, JACM, Vol. 14, No. 4, (Oct.,1967), 687-697.

10. Weissman, C.: LISP 1.5 Primer, Dickenson Publishing Company, Belmont, California (1967).

11. Robinson, G. and Wos, L.: Paramodulation and Theorem-Proving in First-Order Theories with Equality, (to appear in Machine Intelligence, Vol.IV, ed. by D. Michie.)